AMENDMENTS TO THE SPECIFICATION

Please insert the following paragraphs after [0064] on page 24 and before [0065] on page 25. Accordingly, Conclusion in paragraph [0065] will be renumbered as [0097].

Exemplary Architectural Between Runtime and Host Memory Abstraction

[0065] Fig. 3 shows an exemplary architectural relationship between runtime and hosting application memory abstraction interfaces. A memory management abstraction of the runtime hosting interfaces allow the host 304 to provide an interface (i.e., one of the exposed host application interfaces (HAIs) 133) through which the runtime 302 will request all memory allocations. In one implementation, the host interface 133 provides methods that replace both operating system memory API's and standard C runtime allocation routines.

[0066] The memory management abstraction of the runtime hosting interfaces may provide a mechanism for the host 304 to abstract the low memory notification the runtime 302 currently gets from the operating system. This provides the host 304 with a mechanism to ask the runtime 302 to make additional memory available, for example, via garbage collection services.

[0067] The memory management abstraction of the runtime hosting interfaces allow the runtime 302 to inform the host 304 of the consequences of failing a particular allocation, and further allow the host 304 to customize the action the runtime 302 should take if an allocation must be failed. For example, should the runtime 302 unload an application domain or let it run in a "crippled" state.

Exemplary Architectural Relationship Between Runtime and Host Interfaces

[0068] Fig. 4 shows an exemplary architectural relationship between runtime and host application thread abstraction interfaces. The four interfaces shown are IRuntime Task Manager, IHost Task Manager, ICLR (Common Language Runtime) Task, and IHost Task. The thread management APIs may allow the host to provide an interface 133 that the runtime may use to create and start new tasks (threads) and may provide the IHost Task Manager with a mechanism to "reuse" or pool, a runtime implemented portion of a task. This allows for performance optimization that may be used by a host application to minimize host-implemented task creation and setup operations.

[0069] The thread management APIs may implement a callback to notify the IRuntime Task Manager, when a task has been moved to or from a runnable state. When a call is moved from a runnable state, the host API allows the runtime to specify that the task should be rescheduled by the host as soon as possible. Furthermore, the thread management API may provide a way for the IRuntime Task Manager to notify the host that a given task cannot be moved to a different physical operating system thread and cannot have its execution blocked during a specified window of time.

[0070] As shown in Fig. 4, the four interfaces comprise the threading abstraction of this implementation:

[0071] • IHostTaskManager. Implemented by the host 132 and discovered by the runtime 130 via IHostControl::GetHostManager. Allows the runtime 130 to work with tasks through the host 132 instead of using standard OS threading or fiber API's. This interface provides methods for the runtime 130 to create and manage tasks, to notify the host 132 that

the locale has been changed through the BCL, to provide hooks for the host 132 to take action when control is transferring in/out of the runtime 130 to run unmanaged code.

[0072] • IRuntimeTaskManager. Hosts can use IRuntimeTaskManager to create runtime tasks explicitly and to get the currently running runtime task.

[0073] • IHostTask. The runtime 130 creates tasks using IHostTaskManager. Every task created through IHostTaskManager has a representation on the host 132 side and on the runtime 130 side. IHostTask is the task's representation on the host 132 side while IRuntimeTask below for the corresponding interface on the runtime 130 side. Once the task has been created, IHostTask allows the runtime 130 to perform various actions on the task including starting it, setting its priority, alerting it, and other similar functions.

[0074] • IRuntimeTask. The runtime's representation of the task. There is always a corresponding instance of an IHostTask on the host 132 side. The host 132 calls IRuntimeTask to notify the runtime 130 about the state of the task, including when the task is moving to or from the runnable state, or when it is being aborted.

[0075] As described above, each task running in a hosted environment has a representation on the host 132 side (an instance of IHostTask) and a representation on the runtime 130 side (an instance of IRuntimeTask). The host 132 task and runtime task objects should be associated with each other in order for the host 132 and the runtime 130 to properly communicate about the underlying task. The two task objects should be created and associated before any managed code can run on the OS thread.

[0076] The process of creating a new task and its associated task objects can be initiated either by the host 132 or by the runtime 130. The runtime 130 will begin the process of creating new tasks when it is first setting up a thread to run managed code. This

occurs during the runtime's initialization process, when a user explicitly creates a thread using the classes in System.Threading or when the size of the thread pool grows.

[0077] In these scenarios, the steps taken to setup the new task include, for example, the following:

[0078] • The runtime 130 will create a runtime task and the corresponding IRuntimeTask internally. Note: runtime tasks can also be "recycled" – instead of creating a new runtime task from scratch, the runtime 130 may choose to grab a task from its cache of recycled tasks. See IRuntimeTask::Reset for details.

[0079] • The runtime 130 will then call IHostTaskManager::CreateTask to create a host 132 task to associate with the new runtime task.

[0080] • The association is made when the runtime calls SetRuntimeTask on the new IHostTask.

Abstraction Interfaces for Entering and Leaving the Runtime

[0081] Fig. 5 shows exemplary managed/unmanaged code call sequences for exiting (leaving)/reverse-exiting and entering/reverse-entering managed code from/to unmanaged code to/from unmanaged code. Threads running managed code may leave the runtime 130 to run unmanaged code. Locks taken on threads that leave the runtime 130 to run unmanaged code will not go through the RHI 131 so they may not be integrated with the threading and synchronization models of the host application 132. As such, the runtime notifies the host application via a host-implemented callback (the callback being provided by the host to the runtime through a corresponding RI 134) when a thread is entering or leaving the runtime 130 respectively to/from unmanaged code. In view of this, the runtime:

[0082] •Notifies the host 132 when a thread transitions into and out of the runtime 130. Such notifications are implemented by hooking calls out-of and into the runtime regardless of whether code has been compiled in a Just-In-Time (JIT) compiling scenario or in a native image compilation scenario (e.g., ngen). In one implementation, the notification includes the address of the routine being called.

[0083] •Allows the host 132 to specify that a particular function call to unmanaged code and corresponding runtime re-entry is not to be hooked by the runtime 130 for such notification. Such host specification allows the runtime to implement the particular call in an optimized manner (e.g., implementing the call inline). Hosts can use this interface to bypass the hook for calls they "know about" (i.e., calls that are either a part of the implementation of the host itself or of tightly integrated functionality).

[0084] For example, such a host-implemented callback allows the runtime 130 to send the host 132 a "notification" (a call to the hook) that tells the host that a particular thread's behavior can no longer be predicted since it has exited the runtime 130 into user code. Responsive to such a notification, the host 132 may take proactive steps to ensure that the particular thread is not scheduled by the host 132 to participate in any non-preemptive scheduling activity particular to the host's specific implementation, until the thread returns to the runtime.

[0085] In one implementation, such a hook may be used by the host 132 to adjust the floating point state of a processor 104. The host 132 may also utilize the RI 134 to indicate that one or more particular function calls are not to be hooked by the runtime 130, for example, to avoid runtime notifications when the host calls a data access API.

Runtime Tasks Treated as Fibers

[0086] Fig. 6 is a block diagram showing exemplary task scheduling of runtime tasks that are treated as fibers scheduled on operating system threads by a host application. The term "task" is often used to define this abstraction and used to decouple the word "thread" from a particular host application's unit of execution and scheduling.

[0087] A thread management API may allow the host to provide an interface that the runtime may use to create and start new tasks, such as an OS Thread 1 and an OS Thread 2. The thread management API may provide the host with a mechanism to "reuse" or pool, a runtime implemented portion of a task. This allows for performance optimization that may be used by a host application to minimize host-implemented task creation and setup operations.

[0088] By way of illustration, a host application implements a "fiber mode" execution. In fiber mode, a particular host (e.g., an SQL server) may create some number of threads based on the number of processors on the computing device, or based on other host-specific criteria. The host then creates fibers on those threads on which to run user code (a portion of "other program modules"). The host schedules these fibers in a cooperative fashion (called non-preemptive in host terminology) - when a fiber blocks for some operation, it gets "switched out" and the thread runs another fiber. Later the fiber will get rescheduled and run—not necessarily on the same thread. When the runtime creates a "task" through the hosting API, it ends up as a fiber in the host and is natively understood by the host's scheduler.

[0089] For example, Fig. 6 shows how the fibers may operate as "SwitchIn" and how the fibers may block for some operation, shown as "SwitchOut". In these instances, the OS Thread1 or OS Thread2 may run another fiber once the fiber becomes blocked, shown as SwitchOut.

[0090] This SwitchIn notifies the runtime 130 that the task is now in the runnable state. A handle to the OS thread that the task has been scheduled on is passed as a parameter. If impersonation has been done on this thread, it should be reverted (RevertToSelf) before switching in the runtime task. SwitchIn cannot be called twice without a corresponding SwitchOut. A thread handle is a handle to the physical thread this task is running on. SwitchOut notifies the runtime 130 that the task has been removed from the runnable state.

[0091] The thread management APIs may allow the host to provide an implementation of the thread pool, providing the runtime with the ability to queue work items, set and query the size of the thread pool, or other types of queuing. In addition, the thread management APIs may provide notifications to the runtime and to the host that a "locale" has been changed on a given task. The locale is related to localization of software. The runtime includes a notion of current locale, and most hosts applications do as well. These notification interfaces allow the runtime and the host to tell each other if the locale has been programmatically changed on either side – so both sides are kept in sync.

[0092] For example, if the locale is changed on the runtime side, that may affect sorting order in a database implemented by the host. In addition, the thread management APIs may allow the runtime to delay host abort of a given task and may provide means for the runtime (and user code) to adjust the priority of a task.

Enter/Leave Methods

[0093] Fig. 7 shows the host 132 interface has two sets of enter/leave methods to differentiate nested calls from their "outer level" calls: Leave/EnterRuntime and ReverseLeave/ReverseEnterRuntime. The series of calls to LeaveRuntime, EnterRuntime, ReverseLeaveRuntime and ReverseEnterRuntime in these scenarios form a "stack" that lets the host 132 identify the nesting layers. By way of illustration, shown is an exemplary full stack of calls that were pushed and later popped between runtime and hosting application thread abstraction interfacing operations. As shown, LeaveRuntime is called when an "outer level" call is about to leave the runtime 130 and enter unmanaged code, shown in Figure 5.

Assembly Loading Abstraction

[0094] Fig. 8 shows the assembly loading abstraction consists of interfaces that allow hosts to customize the assembly loading process. Specifically, hosts can supply a list of assemblies that should be loaded domain-neutral and customize the way assemblies are located and loaded. The interfaces in the Assembly Loading Abstraction are:

[0095] IHostAssemblyManager. The runtime 130 asks for this top level interface through IHostControl::GetHostManager when the runtime 130 is initialized. If an implementation of this interface is provided, it is assumed that the host 132 wishes to control some aspect of the assembly binding process. IHostAssemblyManager contains methods for the host 132 to provide the list of assemblies that should be loaded domain-neutral, the list of assemblies to which runtime 130 should bind, and to supply an implementation of

IHostAssemblyStore through which the host 132 can implement their own custom assembly resolution process.

[0096] IHostAssemblyStore. To load an assembly from somewhere other than the file system, a host 132 typically catches an AssemblyResolve event on System.AppDomain and provides a byte array containing the assembly's bits. An implementation of IHostAssemblyStore provides additional host-specific stores from which it can bind. If an IHostAssemblyStore is provided, runtime 130 will call back to the host 132 through this interface when binding to an assembly. The host 132 is free to load the assembly from anywhere it chooses and with whatever rules it deems appropriate. In essence, hosts can use IHostAssemblyStore to completely "bypass" Runtime 130 if so desired.